# Assignment 3. Recursion!



## Due Friday, February 3 at 1:00 pm

Each student begins with four late days that may be used throughout the quarter. You may submit this assignment 24 hours late by using one late day or 48 hours late using two late days. No submissions will be accepted more than 48 hours after the due date without prior approval by the head TA. See the syllabus for more information about our late policies.

All due dates and submission times are expressed in Pacific time.

***You are permitted to work on this assignment in pairs.***

This assignment is all about recursive problem-solving. You've practiced writing recursive functions in Assignment 1 and in section, and now it's time to take those skills, combine them with the techniques we've covered over the past couple of lectures, and make some pretty impressive pieces of software.

We've chosen these problems because we think they're a great sampler of the different sorts of fundamental recursive techniques that we've explored. We hope that you find these problems interesting and get a better feel for what recursion can do!

This assignment has one debugger exercise and three coding components, and you have seven days to complete it. We recommend that you start this assignment early and make slow, steady progress throughout the week. Here's a recommended timetable:

- Complete Exploring the Towers of Hanoi the day this assignment goes out.
- Complete Human Pyramids within three days of this assignment going out.
- Complete Protein Synthesis within four days of this assignment going out.
- Complete Inverse Genetics within seven days of this assignment going out.

Recursive problem-solving can take a bit of time to get used to, and that's perfectly normal! It's a new way of thinking about problem-solving and is a bit of an adjustment from traditional iterative programming. Starting this assignment early and making slow, steady progress is a great way to give yourself time to internalize recursive problem-solving.

As always, feel free to get in touch with us if you need any assistance.

We're happy to help out!

You are welcome to work in pairs on this assignment. If you do, please note that

> ***it is not a good idea to have one partner do two of the four parts of this assignment and to have the other partner do the other two,*** that
>
> ***you are responsible for understanding how to solve all of the problems given out here, regardless of how you divvy up the work,*** and that
>
> ***if you are working in pairs, both partners should be looking at the same screen the whole time the assignment is being worked on.***

Historically, students that have split the work in half have had substantially below-average learning outcomes in the course, especially when it comes to the exams. On the other hand, students that sit at the same computer and talk through what they're doing with their partner tend to have substantially above-average learning outcomes in the course.

Stated succinctly, ***pair programming is about working together, not working separately***.

# Assignment Logistics

## Starter Files

We provide a ZIP of the starter project. Download the zip, extract the files, and double-click the `.pro` file to open the project in Qt Creator.

📦 Starter code

## Resources

Feel free to check out our Python-to-C++ guide if you're moving from Python to C++. Also, check out our style guide, guide to testing, and debugging guide.

## Getting Help

Keep an eye on the Ed forum for an announcement of the Assignment 3 **YEAH** (YEAH = Your Early Assignment Help) group session where our veteran section leaders will answer your questions and share pro tips. We know it can be daunting to sit down and break the barrier of starting on a substantial programming assignment – come to YEAH for advice and confidence to get you on your way!

We also here to help if you get run into issues along the way! The Ed forum is open 24/7 for general discussion about the assignment, lecture topics, the C++ language, using Qt, and more. Always start by searching first to see if your question has already been asked and answered before making a new post.

To troubleshoot a problem with your specific code, your best bet is to bring it to the LaIR helper hours or office hours.

# Part One: Explore the Towers of Hanoi

When you encounter a bug in a program, your immediate instinct is probably to say something like

> *"Why isn't my program doing what I want it to do?"*

One of the best ways to answer that question is to instead answer this other one:

> *"What **is** my program doing, and why is that different than what I intended?"*

The debugger is powerful tool for answering questions like these. You explored the debugger in Assignment 0 (when you learned how to set breakpoints and Step Over) and in Assignment 1 (when you learned how to walk up and down the call stack). This part of the assignment will refresh these skills and introduce two other ways to walk through your program in the debugger: ***Step In*** and ***Step Out***. We'll do so in the context of the ***Towers of Hanoi problem***, a classic puzzle that has a beautiful recursive solution. If you haven't yet done so, take a few minutes to read Chapter 8.1 of the textbook, which explores this problem in depth.

Let's begin our exploration. We've provided you with a `TowersOfHanoi.cpp` file, which includes a correct, working solution to the Towers of Hanoi problem. To familiarize yourself with what the starter program does, take a minute to run the "Towers of Hanoi" demo from the main program. Choose the "Go!" button to begin the animation, and marvel at how that tiny recursive function is capable of doing so much. Isn't that amazing?

Now, let's bust out the debugger. Open the `TowersOfHanoi.cpp` file, then set a breakpoint on the first line of the `solveTowersOfHanoi` function. Run the program in debug mode, choose the "Towers of Hanoi" option from the menu at the top of the program, but don't click the "Go!" button yet. When the debugger engages, it halts execution of the running program so that you can inspect what's going on. This means that the graphics window might not be operational – you might find that you can't drag it around, or resize it, or move it, etc. Therefore, we recommend that before you hit the "Go!" button to bring up the debugger, you resize the demo app window and the Qt Creator window so that they're both fully visible.

Once you're ready, hit the "Go!" button. This will trigger the breakpoint. You'll see a yellow arrow pointing at the line containing the breakpoint, and the local variables window will have popped up.

First, investigate the pane in the debugger that shows local variables and their values. Because `totalMoves` has not yet been initialized, its value is unspecified; it might be 0, or it might be a random garbage value. The function's parameters, though, should be clearly visible at this point.

You should now be able to answer the following questions. To do so, edit the file `DebuggingAnswers.txt` with your answers:

### Question 1

What are the values of all the parameters to the `solveTowersOfHanoi` function?

### Question 2

Some function in our starter code called `solveTowersOfHanoi`. What file was that function defined in, and what was the name of that function? (Hint: use the call stack!)

Once you've answered these questions, go back to the `TowersOfHanoi.cpp` file, and make sure you see a yellow arrow pointing at the line containing your breakpoint. Let's now single-step through the program. As a reminder, the "Step Over" button that you saw in Assignment 0 tells the debugger to execute the current line of code in the program. Each time you click it, the program will run the current line and move on to the next. Remember that there's no way to "rewind" the program back to where it was before; think of "Step Over" as a way of hitting "unpause" followed by a quick "pause" once you hit the next line.

Use the "Step Over" button to advance past the call to the function `initHanoiDisplay`, which configures the graphics window. If you've done this correctly, you should see the disks and spindles.

Now, keeping clicking "Step Over" to advance through the other lines in the function. When you step over the line containing the call to `moveTower`, you should see the disks move to solve the Towers of Hanoi. Doesn't get old, does it? 😃

You should now be ready to answer the following question in `DebuggingAnswers.txt`.

### Question 3

How many total moves were required to solve this instance of Towers of Hanoi?

At this point, hit the "Continue" button to let the program keep running as usual. Click the "Go!" button again to trigger your breakpoint a second time.

This time, instead of using Step Over, we're going to use ***Step Into***. Rather than stepping over function calls, Step Into goes inside the function being called so you can step through each of its statements. (If the current line is not a function call, Step Into and Step Over do the same thing.)

Use Step Into to enter the call to `initHanoiDisplay`. The editor pane will switch to show the contents of the `src/Demos/TowersOfHanoiGUI.cpp` file and the yellow arrow will point to the first line of the `initHanoiDisplay` function. This code is unfamiliar, you didn't write it, and you didn't intend to start tracing it. ***Step Out*** is your escape hatch. This "giant step" executes the rest of the current function up to where it returns. Use Step Out to return to `solveTowersOfHanoi`.

The next line of code in `solveTowersOfHanoi` is the pause function, another library function you don't want to trace through. You could step in and back out, but it's simpler to just Step Over.

You are interested in tracing through the **moveTower** function, so use Step Into to go inside. Once inside, single-step through the code until the program is just about to execute the first recursive call to **moveTower**. Now, press Step Over to execute it. The GUI window will show the left tower, except for the bottom disc, moving from the left peg to the middle peg, leaving the bottom disk uncovered. This should also cause the value of **movesOne** to count all moves made by that recursive call. Now, answer the following question:

### Question 4

What is the value of the **movesOne** variable inside the first **moveTower** call after stepping over its first recursive sub-call? (In other words, just after stepping over the first recursive sub-call to **moveTower** inside the **else** statement in the recursive function.)

The next Step Over moves the bottom disk. The final Step Over moves the smaller tower on top. Use Continue to resume normal execution and finish the demo.

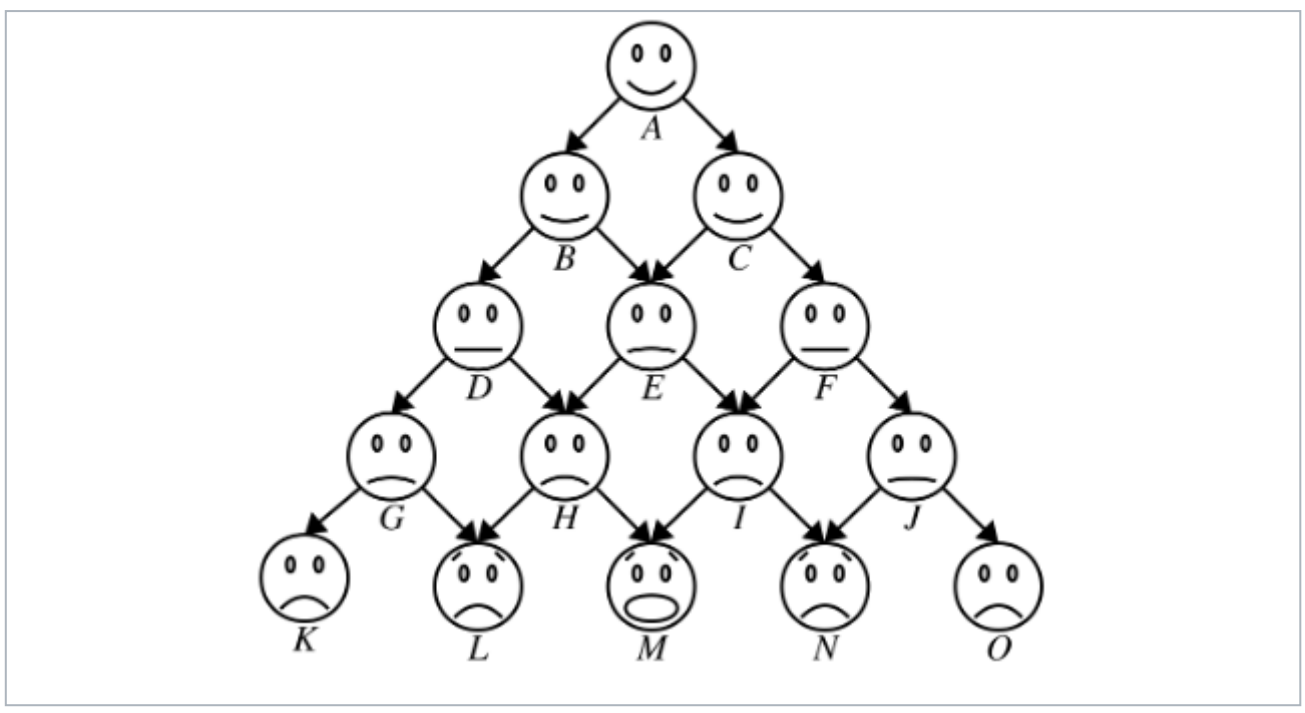
Press the "Go!" button a third time. This time, do your own tracing and exploration to solidify your understanding of recursion and its mechanics. Watch the animated disks and consider how this relates to the sequence of recursive calls. Observe how stack frames are added and removed from the debugger call stack. Select different levels on the call stack to see the value of the parameters and the nesting of recursive calls. Here are some suggestions for how stepping can help:

- Stepping **over** a recursive call can be helpful when thinking holistically. A recursive call is simply a "magic" black box that completely handles the smaller subproblem.
- Stepping **into** a recursive call allows you to trace the nitty-gritty details of moving from an outer recursive call to the inner call.
- Stepping **out** of a recursive call allows you to follow along with the action when backtracking from an inner recursive call to the outer one.

# Part Two: Human Pyramids

A ***human pyramid*** is a way of stacking people vertically. With the exception of the people in the bottom row, each person splits their weight evenly on the two people below them in the pyramid.

For example, consider the following human pyramid:




In the pyramid shown above, person $A$ splits her weight across people $B$ and $C$, and person $H$ splits his weight – plus the accumulated weight of the people he's supporting – onto people $L$ and $M$. It can be mighty uncomfortable to be in the bottom row, since you'll have a lot of weight on your back! In this question, you'll explore just how much weight that is. Just so we have nice round numbers here, let's assume that everyone in the pyramid weighs exactly 160 pounds.

Person $A$ at the top of the pyramid has no weight on her back. People $B$ and $C$ each carry half of person $A$'s weight, so each shoulders 80 pounds. Uncomfortable, but not too bad.

Now, let's consider the people in the third row. Focus on person $E$. How much weight is she supporting? She is supporting person $B$ and person $C$, so she is carrying

- half of person $B$'s body weight (80 pounds), plus
- half of person $C$'s body weight (80 pounds), plus
- half of the weight person $B$ is carrying (40 pounds), plus
- half of the weight person $C$ is carrying (40 points).

That works out to a total of 240 pounds - a noticeable amount.

Not everyone in that third row is feeling the same amount, though; let's take person $D$ for example. The only weight on person $D$ comes from person $B$ in the row above them to their right, because no one is sitting above their left shoulder. Person $D$ therefore ends up supporting

- half of person $B$'s body weight (80 pounds), plus
- half of the weight person $B$ is holding up (40 pounds),

so person $D$ ends up supporting 120 pounds, only half of what $E$ is feeling! Going deeper in the pyramid, how much weight is person $H$ feeling? Well, person $H$ is supporting

- half of person $D$'s body weight (80 pounds),
- half of person $E$'s body weight (80 pounds), plus
- half of the weight person $D$ is holding up (60 pounds), plus
- half of the weight person $E$ is holding up (120 pounds).

The net effect is that person $H$ is carrying 340 pounds – ouch! A similar calculation shows that person $I$ is also carrying 340 pounds – can you see why? Compare this to person $G$. Person $G$ is supporting

- half of person $D$'s body weight (80 pounds), plus
- half of the weight person $D$ is holding up (60 pounds),

for a net total of 140 pounds. That's a lot, but it's not nearly as bad as what person $H$ is feeling! Finally, let's look at poor person $M$ in the middle of the bottom row. How is he doing? Well, he's supporting

- half of person $H$'s body weight (80 pounds),
- half of person $I$'s body weight (80 pounds),
- half of the weight person $H$ is holding up (170 pounds), and
- half of the weight person $I$ is holding up (170 pounds),

for a net total of 500 pounds. Yikes! No wonder he looks so unhappy.

There's a nice, general pattern here that lets us compute how much weight is on each person's back:

- Each person weighs exactly 160 pounds.
- Each person supports half the body weight of each of the people immediately above them, plus half of the weight that each of those people are supporting.

The question then becomes – given some person in the pyramid, how much weight are they supporting?

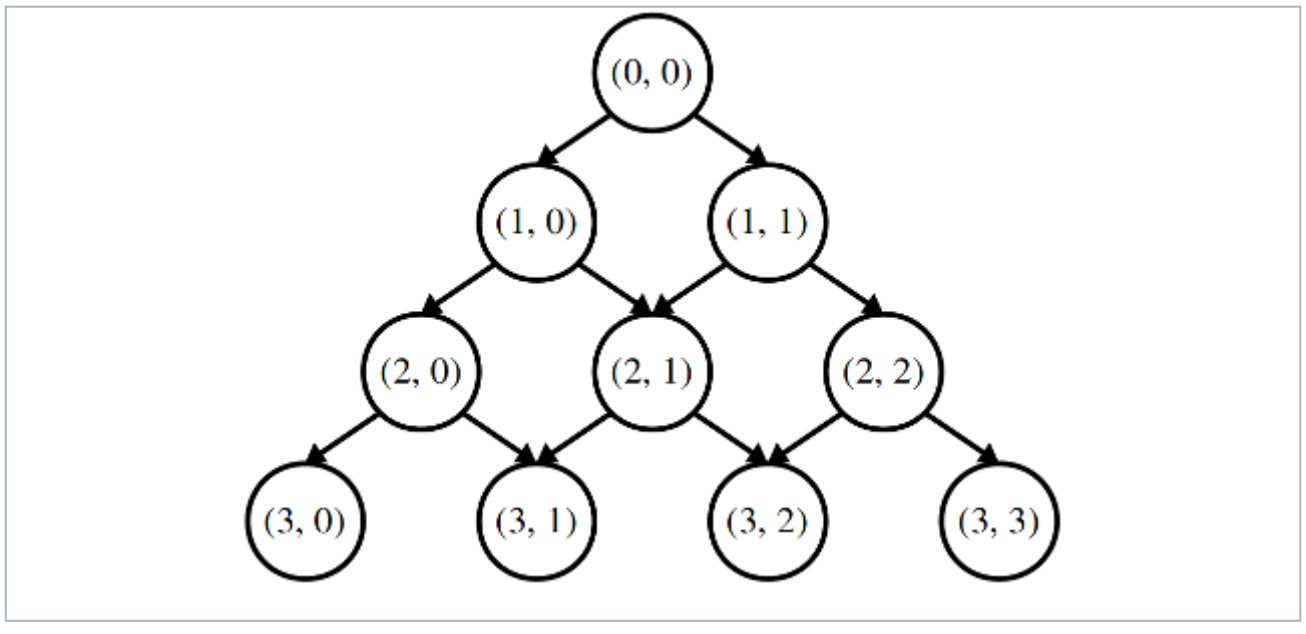
## Milestone 1: The Initial Implementation

Using the pattern mentioned above, as your first milestone for this problem, write a recursive function

```
double weightOnBackOf(int row, int col);
```

that takes as input the row and column number of a person in a human pyramid, then returns the total weight on that person's back.

The coordinate system we're using for these pyramids works as follows. Rows increase from top to bottom. The topmost row is row 0, the row below that is row 1, then row 2, etc. Columns increase from left to right, with the leftmost person in each row being in column 0, the person to their right being in column 1, etc.






In our previous example, person *A* would be at position (0, 0) since she's at the top of the pyramid. Person *B* would be at position (1, 0) since she's the leftmost person in the next row down, and so on and so forth, until Person *O*, who is at position (4, 4). Recall that in our previous pyramid there were 5 rows and the bottommost row has 5 people in it, people *K* through *O*.

If the provided (row, col) position passed into **`weightOnBackOf`** is out of bounds you should use the **`error()`** function to report an error. For example, if you're asked to look up position (-1, 0), or position (2, 3), you'd report an error.

### Milestone 1 Requirements

1. Add at least one test to **`HumanPyramids.cpp`** to cover a case not tested by the other test cases. This is a great opportunity to check that you understand the problem setup.

2. Implement the **`weightOnBackOf`** function from **`HumanPyramids.cpp`** to report the weight on the back of the indicated person in a human pyramid. Test your solution thoroughly before proceeding, noting that your implementation will be slow if you look deep in the pyramid. Don't forget to call **`error()`** if the input coordinates are out of bounds! You should pass all the tests for Milestone 1, but will not yet pass tests associated with Milestone 2.

Some notes on this first milestone:

- You must implement **`weightOnBackOf`** recursively.

- Remember that the **`int`** and **`double`** types are distinct in C++, and be careful not to mix them up. Your weight calculations should be done with **`double`**s. In particular, remember that in C++ the expression `1 / 2` evaluates to 0 because `1` and `2` are **`int`**s. (However, `1.0 / 2.0` evaluates to `0.5`.)

- There are many ways to code this one up. Some of these approaches involve lots of special-case handling. Others don't require much special-case handling at all.

## Milestone 2: Speeding Things Up

When you first code up this function, you'll find that it's pretty quick to tell you how much weight is on the back of the person in row 5, column 3, but that it takes a long time to say how much weight is on the back of the person in row 30, column 15. Why is this?

Think about what happens if we call **`weightOnBackOf(30, 15)`**. This makes two new recursive calls: one to **`weightOnBackOf(29, 14)`**, and one to **`weightOnBackOf(29, 15)`**. This first recursive call in turn fires off two more: one to **`weightOnBackOf(28, 13)`** and another to **`weightOnBackOf(28, 14)`**. The second recursive call then calls **`weightOnBackOf(28, 14)`** and **`weightOnBackOf(28, 15)`**.

Notice that there are two calls to **`weightOnBackOf(28, 14)`** here. This means that there's a redundant call being made to **`weightOnBackOf(28, 14)`**, so all the work done to compute that intermediate answer is done twice. That call will in turn fire off its own redundant

recursive calls, which in turn fire off their own redundant calls, etc. This might not seem like much, but the number of recursive calls can be huge. For example, calling `weightOnBackOf(30, 15)` makes over 600,000,000 recursive calls!

There are many techniques for eliminating redundant calls. One common approach is ***memoization*** (no, that's not a typo). Intuitively, memoization works by making an auxiliary table keeping track of all the recursive calls that have been made before and what value was returned for each of them. Then, whenever a recursive call is made, the function first checks the table before doing any work. If the the recursive call has already been made in the past, the function just returns that stored value. This prevents values from being computed multiple times, which can dramatically speed things up!

In pseudocode, memoization looks something like this:

**Before:**

```
Ret function(Arg a) {
   if (Base-Case-Holds) {
      return Base-Case-Value;
   } else {
      Do-Some-Work;
      return Recursive-Step-Value;
   }
}
```

**After:**

```
Ret functionRec(Arg a, Table& table) {
   if (Base-Case-Holds) {
      return Base-Case-Value;
   } else if (table contains a) {
      return table[a];
   } else {
      Do-Some-Work;
      table[a] = Recursive-Step-Value;
      return table[a];
   }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

In the above pseudocode, we're making reference to a type called `Table`. There isn't actually a type `Table`; rather, it's a placeholder for "some type that you can use to look up a value associated with the arguments to the function." There are many choices you can make here that work well, but we recommend using a `Grid<double>` where each (row, column) pair in the table corresponds to a (row, column) position in the pyramid.

Notice that this transformation makes what was the original function a wrapper function. That's necessary both because we need to create the table somewhere and because we don't want folks using the function to see that the extra parameters have been added in.

To summarize what you need to do:

### Milestone 2 Requirements

1. Take your existing `weightOnBackOf` function and modify it so that it becomes a pair of functions - a wrapper function, plus the actual recursive function. Make sure that the recursive function calls itself rather than the wrapper function.

2. Modify the wrapper function so that it creates a memoization table that it then passes into the recursive function.

3. Modify the recursive function so that it accepts the memoization table and, using the strategy outlined above, avoids recomputing answers that have already been computed.

4. Test your solution thoroughly using our provided tests, plus any others you add.

Some notes on this milestone:

- Make sure that the actual recursive function calls itself and not the wrapper function. If the recursive function calls the wrapper function, it'll discard the memoization table and start with a new one (do you see why?), which will eliminate all the benefits. More generally, the wrapper should always call the recursive function, and the recursive function should never call the wrapper.

- We've included a stress test to make sure that memoization is working correctly. Because it can take a lot of CPU time if memoization isn't implemented correctly, we've programmed the test to always fail until you explicitly turn the test on. Follow the instructions given in the test failure to enable the test.

- There are many different types you can choose for a memoization table, but, as we mentioned above, we recommend choosing `Grid<double>`.

Once you've finished this milestone, choose the "Human Pyramids" option from the menu bar at the top of the program and drag the slider around to change the height of the pyramid. What numbers do you get back? Does that surprise you?

# Part Three: Protein Synthesis

## Background: RNA and Proteins

***RNA*** (ribonucleic acid) is a molecule that encodes genetic information. Plant and animal cells use RNA for a variety of cell functions, while viruses often use RNA as their primary genetic storage. Recently, mRNA, a type of RNA, has been catapulted into public discourse because of its use in COVID-19 vaccines.

Each strand of RNA consists of a series of nucleotides – adenine (`A`), guanine (`G`), uracil (`U`), and cytosine (`C`) – and a strand of RNA can be thought of as a string that only uses those four letters (e.g. `"GUACGUACGUAG"` or `"CAGUACACGUAUC"`).

RNA is used by a cell to encode ***proteins***, biological molecules essential to cell function. Each protein consists of a series of amino acids that, strung together, serve some complex purpose. Similarly to how RNA molecules are strings of nucleotides, proteins are strings of ***amino acids***. There are 22 relevant amino acids for our purposes, and biologists have assigned each of them a letter of the alphabet. For the purposes of this assignment you won't need to know what those letters are, but if you're curious here's the table:

- `A`: Alanine
- `C`: Cysteine
- `D`: Asparitic Acid
- `E`: Glutamic Acid
- `F`: Phenylalanine
- `G`: Glycine
- `H`: Histidine
- `I`: Isoleucine
- `K`: Lysine
- `L`: Leucine
- `M`: Methionine
- `N`: Asparagine
- `O`: Pyrrolysine
- `P`: Proline
- `Q`: Glutamine
- `R`: Arginine
- `S`: Serine
- `T`: Threonine
- `U`: Selenocysteine
- `V`: Valine
- `W`: Tryptophan
- `Y`: Tyrosine

So, for example, the string `"CPP"` would represent a protein consisting of the amino acids cysteine, then proline, then proline, and the string `"KUDU"` would represent a protein consisting of the amino acids lysine, then selenocysteine, then asparitic acid, then selenocysteine.

Here's a very, *very* simplified model of how to translate from an RNA strand to a protein. Start by breaking the RNA strand apart into units that are three characters long. These are called ***codons***. For example, the RNA strand

```
GACAUAAAAGAUAUCAAG
```

would be treated at this sequence of three-letter codons:

```
GAC AUA AAA GAU AUC AAG
```

As the name suggests, each codon "codes" for a particular amino acid. Below is the mapping from codons (three-letter RNA sequences) to proteins (encoded by single letters):

- `AAA` → K
- `AAC` → N
- `AAG` → K
- `AAU` → N
- `ACA` → T
- `ACC` → T
- `ACG` → T
- `ACU` → T
- `AGA` → R
- `AGC` → S
- `AGG` → R
- `AGU` → S
- `AUA` → I
- `AUC` → I
- `AUG` → M
- `AUU` → I
- `CAA` → Q
- `CAC` → H
- `CAG` → Q
- `CAU` → H
- `CCA` → P
- `CCC` → P
- `CCG` → P
- `CCU` → P
- `CGA` → R
- `CGC` → R
- `CGG` → R
- `CGU` → R
- `CUA` → L
- `CUC` → L
- `CUG` → L
- `CUU` → L
- `GAA` → E
- `GAC` → D
- `GAG` → E
- `GAU` → D
- `GCA` → A
- `GCC` → A
- `GCG` → A
- `GCU` → A
- `GGA` → G
- `GGC` → G
- `GGG` → G
- `GGU` → G
- `GUA` → V
- `GUC` → V
- `GUG` → V
- `GUU` → V
- `UAC` → Y
- `UAG` → O
- `UAU` → Y
- `UCA` → S
- `UCC` → S
- `UCG` → S
- `UCU` → S
- `UGA` → U
- `UGC` → C
- `UGG` → W
- `UGU` → C
- `UUA` → L
- `UUC` → F
- `UUG` → L
- `UUU` → F

So, for example, our RNA strand from above would encode this amino acid sequence:

```
GAC AUA AAA GAU AUC AAG
 D   I   K   D   I   K
```

Similarly, consider this RNA strand:

```
CAGUGAUAGAAGAAGGCU
```

It breaks apart into codons like this:

```
CAG UGA UAG AAG AAG GCU
```

And those codons encode the following string:

```
CAG UGA UAG AAG AAG GCU
 Q   U   O   K   K   A
```

## Your Task

Write a recursive function

```
string toProtein(const string& rna, const Map<string, char>& codonMap);
```

that takes in two inputs. The first, **`rna`**, is an RNA strand. The second, **`codonMap`**, is a **`Map<string, char>`** where each key represents a possible codon and each value is the letter of the amino acid it corresponds to. Your function should then return the protein that the RNA strand corresponds to.

You will need to validate the input to make sure it's correct. Specifically, if the RNA strand can't be broken apart into codons (e.g. it has length five), or if you find a group of three letters in **`rna`** that aren't a key in **`codonMap`**, your function should call **`error`** to report an error.

After implementing your function, add at least one custom test case to **ProteinSynthesis.cpp** to make sure that your code works as intended. Read over the tests we've provided and see if you can find a case we didn't consider.

More specifically, here's what you need to do:

### Protein Synthesis Requirements

1. Implement the **toProtein** function in **ProteinSynthesis.cpp**.
2. Add at least one custom **STUDENT_TEST** to **ProteinSynthesis.cpp** to test your **toProtein** function. (And, ideally, add more than just one!)

Some notes on this problem:

- Your function must be implemented recursively.
- You can assume that strings are case-sensitive, so **guac** and **GUAC** are different RNA strands (and, similarly, **gua** and **GUA** are considered different codons).
- The call stack has limited space, so if you call your function on a sufficiently long string, you might get a stack overflow even if you've implemented everything correctly. For the purposes of this assignment, you can assume that no one is going to call your function on an input that's so large that this would happen. (Conversely, though, if you see a stack overflow during testing, that likely indicates a bug in your code rather than an input that's too long.)

# Part Four: Inverse Genetics

If you look at the table mapping codons to amino acids, you'll notice that many codons represent the exact same amino acids. This means that there can be many different RNA strands that all encode the exact same protein. For example, here are a few other RNA strands that represent the amino acid sequence **DIKDIK**:

- **GACAUAAAAGACAUAAAA**
- **GACAUAAAGGACAUAAAG**
- **GAUAUAAAAGAUAUAAAA**
- **GAUAUUAAAGACAUUAAA**
- **GAUAUCAAGGAUAUUAAG**
- **GACAUCAAAGAUAUCAAG**

In total, there are 144 different ways to encode **DIKDIK**. There's 41,472 ways to encode **CPLUSPLUS**, 4,608 ways to encode **STANFORD**, and 36,864 to encode **COVIDVACCINE**.

Write a function

```
Set<string> allRNAStrandsFor(const string& protein,
                             const Map<string, char>& codonMap);
```

that takes as input a protein, represented as a string of the amino acids that it's comprised of, then returns a **Set** of all the RNA sequences that generate it.

This is a recursive enumeration problem and will require you to combine loops and recursion together. You do not need to write a lot of code here, but you will need to be intentional with how you proceed. Here are some hints to get you thinking along the right lines:

1. Given an amino acid, there can be many codons that produce it. Use recursion to explore each possible next codon to see what you find.
2. Draw out a decision tree before you start coding anything up. The decision tree here won't exactly match subsets or permutations, but many of the insights from those problems will still be applicable here.
3. Think about the general pattern we've seen: keep track of what decisions you have made so far and what decisions you haven't made yet.
4. The **Map** you're given to this problem goes from codons to amino acids. However, it might be easier to work with the data if it went from amino acids to codons.

We have provided you with a good range of test cases, but we haven't covered everything. You will need to write at least one custom test case. We recommend that, for testing, you substitute the full RNA-to-codon coding map for something much, much smaller and simpler. That will allow you to focus your tests more precisely.

To summarize, here's what you need to do:

## Inverse Genetics Requirements

1. Implement the function **`allRNAStrandsFor`** in **`InverseGenetics.cpp`** so that it returns all RNA sequences that code for the given protein.
2. Add at least one test case in to the list of test cases, then test thoroughly.
3. After your code passes all the tests, play around with the demo app to have some fun with your program.

Some notes on this problem:

- If you aren't fully sure how the lecture code for subsets or permutations works, ask us questions! The code you need to write here is heavily based on the same intuitions that gave rise to the subsets and permutations code from class. Feel free to ask questions in office hours, at the LaIR, or over EdStem. Building an understanding of how subsets and permutations work will make this problem a lot easier.
- Your solution to this problem must use recursion. That being said, remember that "using recursion" doesn't mean "not using loops," and we encourage you to use loops as you see fit.
- If one or more of the characters of **`protein`** isn't an amino acid described **`codonMap`**, then you should return an empty **`Set`** because there are no ways of encoding the protein. That being said, chances are that you will not need to write any code that explicitly addresses this case. (In fact, if you do have code that addresses this case, see if you can remove it!)
- There can be a *lot* of different ways of representing a protein. For example, there are 28,179,280,429,056 ways to encode **`TURNINGANDTURNINGINAWIDENINGGYRE`**. That's way more than can fit into your computer's memory! You don't need to worry about this; we won't test your code on any inputs where the set of options will exceed, say, 10,000,000.
- Remember to choose the absolute simplest base case possible. Look at the examples of subsets and permutations from lecture as an example of what that might look like.

# (Optional) Part Five: Extensions!

As always, if you'd like to go above and beyond what's required for this assignment, you are welcome to do so in whatever way speaks the most to you. If you're planning on doing extensions, please submit two versions of your assignment – one with exten-sions and one without – so that grading goes more smoothly.

Here are a few suggestions to help you get started.

- ***Human Pyramids:*** There's another way to avoid making multiple recursive calls called *dynamic programming* that's equivalent to memoization, but uses iteration rather that re-cursion. Look up dynamic programming and try coding this function up both ways. Which one do you find easier?

  Another question to consider: We assumed everyone in the pyramid weighed exactly 160 pounds. What if that's not the case? How much weight does everyone feel then?

- ***Protein Synthesis:*** Our model of protein synthesis is greatly simplified from how the process works in nature. We ignore start and stop codons, or context cues that determine whether stop codons code for selenocysteine. Make the code more accurately model how RNA transcription works in nature.

- ***Inverse Genetics:*** The original generation of mRNA COVID vaccines worked by generating mRNA sequences coding for the coronavirus spike protein, and the teams that synthesized this mRNA made several design choices in selecting which strands to use. Do some background reading on ways to select between different seemingly equivalent RNA strands for a protein, then update your code to prioritize "better" strands over "worse" ones.

# General Notes

Here's some clarifications, notifications, expectations, and recommendations for this assignment.

- ***Your functions must actually use recursion;*** it defeats the point of the assignment to solve these problems iteratively! That being said, as you'll see in lecture next week, it's perfectly alright for recursive functions to contain loops. The main question is whether your solution fundamentally works by breaking problems down into smaller copies of themselves. If so, great! If not, you may want to revisit your solution.

- ***Test your code thoroughly!*** We've included some test cases with the starter files, but they aren't exhaustive. Be sure to add tests as you go!

- ***Recursion can take some time to get used to,*** so don't be dismayed if you can't immediately sit down and solve these problems. We've allowed you to work in pairs on this assignment so that you can discuss ideas with a partner, which can be a great way build an intuition for the concepts. Ask for advice and guidance if you need it. Once everything clicks, you'll have a much deeper understanding of just how cool a technique this is. We're here to help you get there!

# Submission Instructions

Before you call it done, run through our submit checklist to be sure all your `t`s are crossed and `i`s are dotted. Make sure your code follows our style guide. Then upload your completed files to Paperless for grading.

**Partner Submissions:**

- If you forget to list your partner you can resubmit to add one
- Either person can list the other, and the submissions (both past and future) will be combined
- Partners are listed per-assignment
- You can't change/remove a partner on an individual submission

Please submit only the files you edited; for this assignment, these files will be:

- `DebuggerAnswers.txt` *(Don't forget this one!)*
- `HumanPyramids.cpp`
- `ProteinSyntheis.cpp`
- `InverseGenetics.cpp`

You don't need to submit any of the other files in the project folder.

Submit to Paperless

If you modified any other files that you modified in the course of coding up your solutions, submit those as well. And that's it! You're done!

***Good luck, and have fun!***